A Dataflow-Based Scientific Workflow Composition Framework

Xubo Fei, *Student Member*, *IEEE*, and Shiyong Lu, *Senior Member*, *IEEE*

Abstract—Scientific workflow has recently become an enabling technology to automate and speed up the scientific discovery process. Although several scientific workflow management systems (SWFMSs) have been developed, a formal scientific workflow composition model in which workflow constructs are fully compositional one with another is still missing. In this paper, we propose a dataflow-based scientific workflow composition framework consisting of 1) a dataflow-based scientific workflow model that separates the declaration of the workflow interface from the definition of its functional body; 2) a set of workflow constructs, including Map, Reduce, Tree, Loop, Conditional, and Curry, which are fully compositional one with another; 3) a dataflow-based exception handling approach to support hierarchical exception propagation and user-defined exception handling. Our workflow composition framework is unique in that workflows are the only operands for composition; in this way, our approach elegantly solves the two-world problem in existing composition frameworks, in which composition needs to deal with both the world of tasks and the world of workflows. The proposed framework is implemented and several case studies are conducted to validate our techniques.

Index Terms—Scientific workflow, scientific workflow model, workflow composition, MapReduce, VIEW.

1 Introduction

Scientific workflow has become a new paradigm for scientists to integrate, structure, and orchestrate heterogeneous and distributed services and applications into scientific processes to enable and accelerate many scientific discoveries. In contrast to business workflows, which focus on the modeling of controlflow oriented business processes, scientific workflows aim to model often large-scale data-intensive and compute-intensive scientific processes. This poses new exciting challenges to scientific workflow management [1] in general and to scientific workflow composition in particular.

In this paper, we argue that there is a great need to design and implement a dataflow-based scientific workflow composition framework in which workflow constructs are fully compositional one with another. First, as more and more scientific research projects use scientific workflow as an enabling technology to automate and speed up the scientific discovery process, productive workflow composition that promotes workflow sharing and reuse becomes increasingly important. Second, while the goal of business workflows is to reduce human resources (and other costs) and increase revenue, the goal of scientific workflows is to reduce both human and computation costs and accelerate the speed of turning large amounts of bits and bytes into knowledge and discovery. As a result, while business workflows are typically controlflow oriented, scientific workflows tend to be dataflow oriented. Therefore, instead of using an existing business workflow language, such as BPEL [2] and YAWL [3], it is highly desirable to have a dataflow-based scientific workflow language to support the specification and execution of complex data-driven scientific workflows. Finally, although several dataflow-based scientific workflow languages have been implemented [4], [5], [6], none of them provides the workflow constructs (e.g., Map and Reduce) that are fully compositional one with another.

Our main contributions are:

1. we identify seven key requirements for a scientific workflow composition model based on a comprehensive literature review and our experience in developing the VIEW system;
2. we propose a new scientific workflow model that separates the declaration of the workflow interface from the definition of its functional body;
3. we then introduce a set of workflow constructs, including Map, Reduce, Tree, Loop, Conditional, and Curry, which are fully compositional one with another;
4. we propose a dataflow-based exception handling approach to support hierarchical exception propagation and user-defined exception handling; and
5. we implement the proposed framework and conduct several case studies to validate our techniques.

2 Key Requirements for a Scientific Workflow Composition Model

Based on a comprehensive study of the workflow literature [7], [8] and our own experience from the development of the VIEW system [9], we identify the following seven key requirements for a scientific workflow composition model.

- *X. Fei is with the Department of Computer Science, Wayne State University, 4757 Anthony Wayne Drive, Apartment 8, Detroit, MI 48202. E-mail: xubo@wayne.edu.*
- *S. Lu is with the Department of Computer Science, Wayne State University, 5143 Cass Avenue, 431 State Hall, Detroit, MI 48202. E-mail: shiyong@wayne.edu.*

Manuscript received 24 Oct. 2009; revised 3 Feb. 2010; accepted 23 June 2010; published online 14 Dec. 2010.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2009-10-0202.
Digital Object Identifier no. 10.1109/TSC.2010.58.

R1: Visual programming-in-the-large. The concepts of "programming-in-the-large" and "programming-in-the-small" were first introduced by DeRemer and Kron in 1976 [10]. While programming-in-the-large focuses on the high-level abstractions of program modules and the specification of their interactions and coordination, programming-in-the-small focuses on the low-level programmatic implementation of modules and functionalities. Given the high-level orchestration and integration nature of scientific workflow composition, a scientific workflow composition model should fall in the programming-in-the-large paradigm. Moreover, since end users are often scientists who will likely work more efficiently with a visual programming environment, we advocate that scientific workflows should adopt the visual programming-in-the-large paradigm [11].

R2: Dataflow programming model. While in an imperative (controlflow-based) programming model, the order of program execution is explicitly specified by controlflow constructs, such as sequential, conditional, and loop, in a dataflow-based programming model, the availability of input data for a module initiates the execution of the module and the movement of data through modules determines the execution order of the whole program. Since most scientific workflows aim at data processing and scientific analysis problems, it is advised that a scientific workflow composition model should be dataflow-based. Although from a user's perspective, controlflow constructs such as Loop and If-Else are important, we show later in this paper that their dataflow-based counterparts are possible. Moreover, a dataflow-based workflow model features implicit parallelism: workflow modules run in parallel by default unless there is an explicit specification that one module needs an input data that is to be produced as the output of another module. Since a dataflow-based programming model [12] eliminates the shared memory assumption and the need for program counter and control sequencer, it can more easily leverage high performance computing (HPC) capabilities enabled by today's variety of parallel and distributed computing infrastructures (Grids, Clouds, multicore, and multiprocessor systems).

R3: Composable dataflow constructs. Current dataflow-based workflow languages are still very basic, providing data links for connecting components, but lacking a rich set of dataflow constructs that are fully compositional one with another. In order to address the requirements of ever-increasing complex e-science applications, some scientific workflow languages borrow controlflow constructs from business workflow languages. However, this results in a hybrid model in which the combination of controlflow and dataflow constructs obscures the semantics of the language, leading to workflows that are hard to analyze and verify for their correctness. In contrast to controlflow constructs, which are used to control and coordinate processes, dataflow constructs are designed for efficient and systematic data processing, including data parallelism and aggregation, recursive data processing with finite or infinite loops, and data-dependent conditional branching. Dataflow constructs should be compositional one with another, so that a second dataflow construct can be applied on a workflow W' obtained by the application of a first dataflow construct on a workflow W. In this paper, we show that the ability to combine basic dataflow constructs into composite ones is an appealing feature for an expressive scientific workflow composition model.

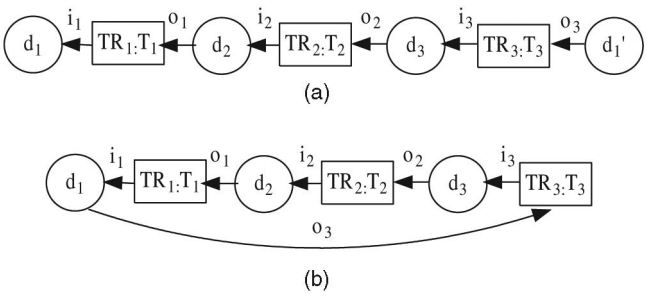


Fig. 1. (a) Correct data dependencies under the single-assignment property. (b) Incorrect data dependencies due to violation of the single-assignment property.

R4: Workflow encapsulation and hierarchical composition. A scientific workflow composition model should facilitate and support workflow encapsulation and hierarchical workflow composition. On one hand, one of the most import features of scientific workflows is to allow the reuse and sharing of scientific processes by workflow encapsulation [13], [14]. A scientific workflow model should provide input/output interfaces while hiding implementation details. Such well-encapsulated modules represent a separation of concerns and improve maintainability. On the other hand, a scientific workflow model should support hierarchical composition so that a user can compose a workflow from existing scientific workflows and break down a large-scale scientific workflow into smaller ones. This ability greatly improves the power of modeling of complex scientific processes and encourages scientific collaborations [15].

R5: Single-assignment property. To ease provenance tracking and workflow scheduling, a scientific workflow composition model should have the *single-assignment property*, in which data products are treated as immutable artifacts; they can be created and transported, but never updated. First, scientific discoveries produced from scientific workflows must be reproducible, requiring the acyclicity of provenance graphs and the immutability of data products [16]. The violation of this property might lead to incorrect data dependencies and thus compromise reproducibility. Fig. 1 illustrates an example of provenance for a workflow consisting of three tasks: T_1 takes input of d_1 and produces d_2; T_2 consumes d_2 and generates d_3; T_3 consumes d_3 and generates d_1' to replace d_1. If the single-assignment property is respected, then d_1' will be a different data product, and we can derive the acyclic dependency graph shown in Fig. 1a: d_1' depends on d_3; d_3 depends on d_2, and d_2 depends on d_1. However, if the single-assignment property is not enforced, then d_1' and d_1 will be treated as the same data product and will be represented by one single node, resulting in a cyclic provenance graph shown in Fig. 1b. Based on transitivity, one can infer that d_2 depends on d_3, a false data dependency relationship. Second, the single-assignment property can eliminate the interference caused by concurrent access (read and write) of data products, which can result in inconsistent and

undesirable intermediate or final results that would not be obtainable if workflow tasks are run in a serial fashion. Third, the single-assignment property can greatly facilitate the realization of massive parallelism: multiple workflow tasks can be started as long as their input data products become available; the single-assignment property ensures the well-defined availability time of each data product; data products can be transported to their consumers directly and removed after consumption without first being stored and then retrieved. As a result, the single-assignment property is assumed by many functional program languages and dataflow programming languages [12]. Finally, unlike business data in business workflows that need to be updated frequently, most scientific data sets are accessed in a read-only manner and updates to data sets are very rare [17]. Therefore, the single-assignment requirement will unlikely have negative impact on the computation and processing of scientific data sets.

R6: Physical and logical data models. Scientific applications usually involve heterogeneous and distributed data [18]. Data management is, thus, becoming one of the key challenges for scientific workflow management [19]. We argue that a scientific workflow composition model should provide a physical data model, a logical data model, and a mapping between them. First, a physical data model is important for the management of distributed data storage, such as local files, databases, and remote files, and heterogeneous physical representations, in which different formats represent the same data. Second, a logical data model should provide data typing and data structures. In order to maintain the integrity and consistency of a scientific workflow composition model, a formal data typing system is required to detect "type errors." Furthermore, data structures with well-defined operators/constructs are also essential for storing and organizing data collections. Third, the separation of the logical data model from the physical data model allows a user to define and manipulate scientific data using the logical data model without the concern of physical data management. As a result, changing of the underlying physical data model will not affect an existing scientific workflow model. Finally, an explicit and standard data mapping layer with precise metadata and explicit data access is necessary to guarantee the efficient and consistent mapping between the physical data model and the logical data model.

R7: Task level and workflow level exception handling. Exceptions in scientific workflows may occur in both the task layer and the workflow layer. A scientific workflow composition model should be able to capture and handle exceptions in both layers. First, while exception handling in business workflows usually focuses on service exceptions such as service failure or deadline expiry [20], [21], [22], scientific workflows may involve heterogeneous tasks (e.g., local executables, grid applications, and cloud services) and exception handling in scientific workflows is, thus, required to be able to detect and integrate heterogeneous exceptions generated by those tasks. Second, because scientific workflows are usually hierarchical and distributed, exception handling in scientific workflows should also be hierarchical and exception propagation should be supported. Third, exceptions in scientific workflows are sometimes very important for scientists to detect hidden problems, improve scientific models and achieve new scientific discoveries. Therefore, despite traditional failure handling techniques, a scientific workflow composition model should allow a user to introduce new exceptions and provide user-defined handlers.

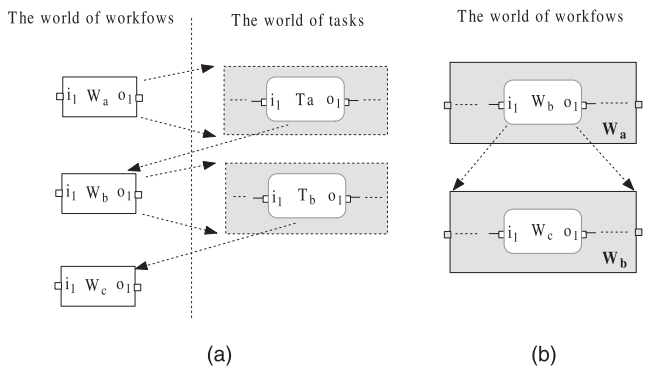


Fig. 2. (a) Traditional scientific workflow model. (b) Our proposed scientific workflow model.

3 Scientific Workflow Model

In current scientific workflow models, a workflow is defined as a composition of tasks that are either primitive or composite. Therefore, these models need to deal with both the world of tasks and the world of workflows. As a result, existing models cannot efficiently support workflow composition. As shown in Fig. 2a, in order to create a three-level hierarchical workflow W_b using existing workflow W_c, first, we need to map W_c to a composite task T_b and then use T_b to compose W_b. Similar mappings will also be needed in order to compose W_a using W_b. Those mappings between workflows and composite tasks are mathematically inelegant and lack mathematical properties to reason about workflow composition. Inspired by functional programming, we propose a new dataflow-based scientific workflow model with a strong functional flavor. As shown in Fig. 2b, workflows are the only operands for composition and the two-world problem is, thus, avoided. In our proposed scientific workflow model, the declaration of the workflow interface is separated from the definition of its functional body. Such a separation provides an abstraction mechanism that makes it possible to introduce dataflow constructs that are fully composable one with another. Specifically, our proposed scientific workflow model consists of the following two layers:

- The *logical layer* contains the *workflow interface* that models the input and output ports of a workflow. The details of the workflow body definition is transparent to this layer.
- The *physical layer* contains the *workflow body* that models the physical implementation of the workflow. Depending on different implementations, a workflow can be either primitive or composite. Primitive workflows are the building blocks of our model with predefined implementations while composite workflows are composed from existing workflows by workflow constructs.

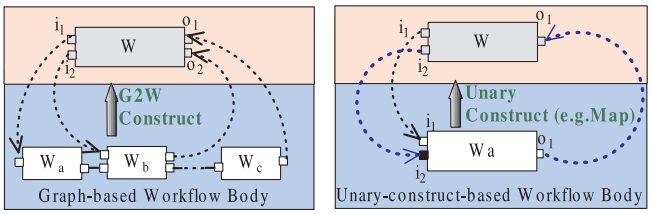


Fig. 3. (a) A graph-based workflow. (b) A unary-construct-based workflow.

Our proposed scientific workflow model is extensible in the sense that future workflow constructs can be easily introduced into the model without affecting the compositionality of existing dataflow constructs. Below we provide a brief overview of three kinds of workflows; but note that such differentiation is made only at the physical layer, not at the logical layer, thus, workflows are uniform objects in our model and they can be composed with each other using various workflow constructs.

Primitive workflows. A set of primitive workflows serve as the basic building blocks of a scientific workflow composition framework. These primitive workflows are the abstractions of heterogeneous and distributed services and applications (tasks) that are dynamically mapped to resources at runtime. A more detailed description of this abstraction and mapping technique can be found in [23]. Our proposed workflow composition framework, however, is orthogonal to how each primitive workflow is built and mapped to resources during execution.

Graph-based workflows. A set of workflows can be connected to each other via their ports through data channels to form a *workflow graph* G. During workflow execution, these workflows communicate with each other by passing data through data channels. As shown in Fig. 3a, the $G2W$ construct is then applied to workflow graph G to construct a graph-based workflow. $G2W$ essentially performs the mapping between the input/output ports of a workflow and the input/output ports of the workflows in its constituent workflow graph, and thus exposes some of the input/output ports of the workflows in G as the input/output ports of the target workflow.

Unary-construct-based workflows. A unary construct U can be seen as a mapping from workflows to workflows. Therefore, given a workflow W, $U(W)$ is another workflow whose behavior depends on both W and U. Unary constructs are very useful to enhance the capability of an existing workflow without coding effort and to promote the reuse of existing workflows in various contexts. For example, our to-be-proposed Map construct can be used to transform a workflow that can only process a single data product to one that can perform the parallel processing of a list of data products. As shown in Fig. 3b, a unary-construct essentially performs the mapping between the input/output ports of a workflow and the input/output ports of its constituent workflow, and carries out the semantics that is defined for the unary construct during runtime (e.g., the map port of the Map construct).

4 Scientific Workflow Constructs and Composition

A unary construct can be applied on a many-inputport-one-outputport workflow. As shown in Fig. 4, six common unary constructs are currently supported by our model: Map, Reduce, Tree, Conditional, Loop, and Curry. More specifically, to apply a unary construct S on W_a, we define a new unary-construct-based workflow $W_b = S(W_a)$. There is an isomorphic mapping between the input/output ports of W_a and W_b. Each corresponding pair of ports have the same type, i.e., $dom(W_a.i_j) = dom(W_b.i_j)(j = 1, \ldots, n)$, and $dom(W_a.o) = dom(W_b.o)$, except for the designated ports specified by the constructs.

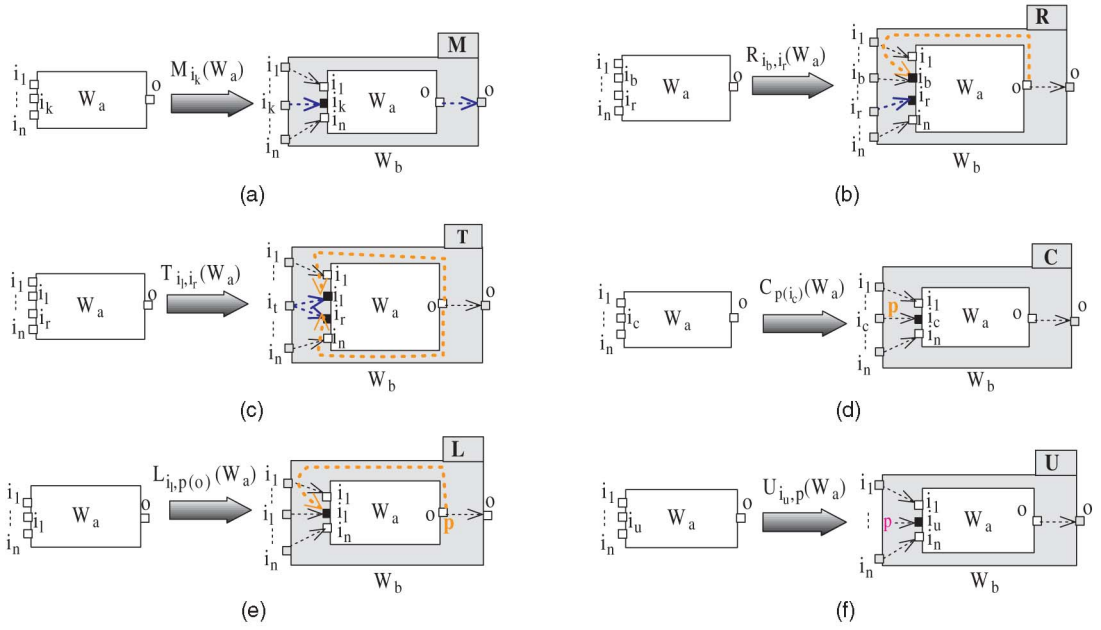


Fig. 4. Six unary workflow constructs. (a) Map. (b) Reduce. (c) Tree. (d) Conditional. (e) Loop. (f) Curry.

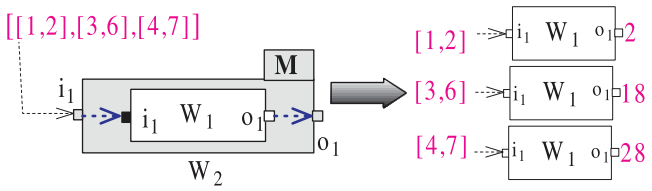


Fig. 5. Workflow W_2 created by applying the Map construct on W_1.

Although the unary constructs can only be applied to workflows with a single output, it is not a limitation. The application of the construct on general workflows can be simulated by the assistance of some primitive workflows for list operations such as *Merge* and *Split* (illustrated by a later example).

In contrast to the original MapReduce model [24], which supports only task-level Map jobs (tasks), our Map and Reduce constructs can be applied to arbitrary scientific workflows; moreover, the original MapReduce model can only process key/value pairs, while our model can process data products of various types. Therefore, our model promotes the power of MapReduce from the task level to the workflow level and enables Map and Reduce fully composable with themselves and with other workflow constructs in both flat and hierarchical manners.

4.1 The Map Construct

The Map construct enables the parallel processing of a list of data products based on a workflow that can only process a single data product. As illustrated in Fig. 4a, given a workflow $W_a([i_1, \ldots, i_n], o)$ with n input ports, $i_1, i_2, \ldots$, and i_n, and one output port o, to apply the Map construct on W_a, one of the input ports of W_a, $i_k \in [i_1, \ldots, i_n]$, is designated as the *map port*, which takes a list of data products that need to be processed in parallel. If $W_a.i_k$ has type T_1, then $W_b.i_k$ has type List of T_1; if $W_a.o$ has type T_2, then $W_b.o$ has type List of T_2. The semantics of the Map construct $M_{i_k}(W_a([i_1, \ldots, i_n], o))$ can be formulated by the following equation:

$$\begin{aligned} W_b.o &= W_b(i_1, \ldots, [i_{k_1}, i_{k_2}, \ldots, i_{k_m}], \ldots, i_n) \\ &= [W_a(i_1, \ldots, i_{k_1}, \ldots, i_n), \ldots, \\ &\quad W_a(i_1, \ldots, i_{k_m}, \ldots, i_n)]. \end{aligned} \tag{1}$$

Our Map construct does not require key-value pairs as in the traditional MapReduce model. Instead, our Map construct takes a list of data products as input and the index of a data product within a list can be considered as a default "key." Our Map construct is order preserving in the sense that each output data product has the same index as that of the corresponding input data product.

For example, Fig. 5 illustrates a workflow W_2 that multiplies each pair of numbers in the input list. Given workflow W_1 that takes a pair of numbers as input and outputs their product, W_2 is created from W_1 by applying the Map construct with the input port i_1 designated as the map port. Given an input list $[[1, 2], [3, 6], [4, 7]]$, the output of W_2 is

$$\begin{aligned} W_2.o &= W_2([[1, 2], [3, 6], [4, 7]]) \\ &= [W_1([1, 2]), W_1([3, 6]), W_1([4, 7])] \\ &= [2, 18, 28]. \end{aligned}$$

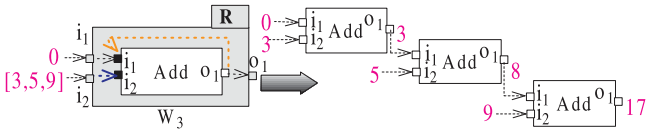


Fig. 6. Workflow W_3 created by applying the Reduce construct on an *Add* Workflow.

4.2 The Reduce Construct

The Reduce construct enables the aggregation of a list of data products to a single data product based on a workflow that aggregates *only* two input data products. As illustrated in Fig. 4b, to apply the Reduce construct on a workflow $W_a([i_1, \ldots, i_n], o)$, an input port, $i_r \in [i_1, \ldots, i_n]$ is designated as the *reduce port*, which takes input from the list of data products that need to be aggregated, another input port, $i_b \in [i_1, \ldots, i_n]$ is designated as the *base port*, which takes input either from an initial base data product or from the intermediate aggregation data product that is produced as the output of the previous iteration of aggregation. If $W_a.i_r$ has type T_1, then $W_b.i_r$ has type List of T_1. Moreover, since port i_r may take input from the previous output, it is required that $dom(W_a.o) \subseteq dom(W_a.i_b)$. The semantics of the Reduce construct $R_{i_b,i_r}(W_a([i_1, \ldots, i_n], o))$ can be formulated by the following equation:

$$\begin{aligned} W_b.o &= W_b(i_1, \ldots, i_b, \ldots, [i_{r_1}, i_{r_2}, \ldots, i_{r_m}], \ldots, i_n) \\ &= o_m, \\ where\ o_1 &= W_a(i_1, \ldots, i_b, \ldots, i_{r_1}, \ldots, i_n), \\ o_2 &= W_a(i_1, \ldots, o_1, \ldots, i_{r_2}, \ldots, i_n), \\ &\ldots, \\ o_m &= W_a(i_1, \ldots, o_{m-1}, \ldots, i_{r_m}, \ldots, i_n). \end{aligned} \tag{2}$$

For example, Fig. 6 illustrates a workflow W_3 that calculates the sum of all the numbers in the input list. W_3 is created from a predefined workflow *Add* by applying the Reduce construct with input ports i_1 and i_2 designated as the base port and the reduce port, respectively. A default value 0 is set on port i_1 as the base value. Given an input list $[3, 5, 9]$, the output of W_3 is

$$\begin{aligned} o_1 &= Add(0, 3) = 3, \\ o_2 &= Add(3, 5) = 8, \\ o_3 &= Add(8, 9) = 17, \\ W_3.o &= W_3(0, [3, 5, 9]) = o_3 = 17. \end{aligned}$$

4.3 The Tree Construct

The Tree construct enables parallel aggregation of a list of data products. In contrast to the Reduce construct, which performs a sequential aggregation, the Tree construct aggregates a list pairwisely as a binary tree until one single aggregated product is generated. As illustrated in Fig. 4c, to apply the Tree construct on a workflow $W_a([i_1, i_2, \ldots, i_n], o)$, two input ports, $i_l, i_r \in [i_1, i_2, \ldots, i_n]$ are designated as the *left tree port* and the *right tree port*. The resulting unary-construct-based workflow will have a corresponding *tree port*, which takes inputs of a list of data products that need to be aggregated. If $W_a.i_l$ and $W_a.i_r$ have type T_1, then $W_b.i_t$ has type List of T_1. The semantics of the Tree construct $T_{i_l,i_r}(W_a([i_1, i_2, \ldots, i_n], o))$ can be formulated by the following recursive equation:

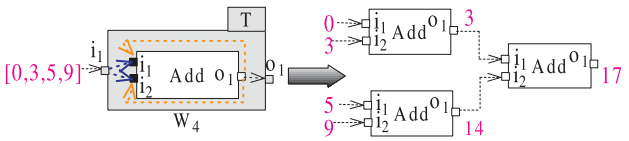


Fig. 7. Workflow W_4 created by applying the Tree construct on an *Add* Workflow.

$$\begin{aligned} W_b.o &= W_b(i_1, \ldots, [i_{r_1}, \ldots, i_{r_m}], \ldots, i_n) \\ &= i_{r_1}, \quad if\ m = 1 \\ &\quad W_a(i_1, \ldots, \\ &\quad W_b(i_1, \ldots, [i_{r_1}, \ldots, i_{r_{\lceil m/2 \rceil}}], \ldots, i_n), \\ &\quad W_b(i_1, \ldots, [i_{r_{\lceil m/2 \rceil + 1}}, \ldots, i_{r_m}], \ldots, i_n), \\ &\quad \ldots, i_n) \qquad if\ \ m > 1. \end{aligned} \tag{3}$$

For example, Fig. 7 illustrates a workflow W_4 that calculates the sum of the input list. W_4 is created from a primitive workflow *Add* by applying the Tree construct with input ports i_1 and i_2 designated as the left tree port and the right tree port, respectively. Given an input list $[0, 3, 5, 9]$, the output of W_3 is

$$\begin{aligned} W_4.o &= W_4([0, 3, 5, 9]) \\ &= Add(Add(0, 3), Add(5, 9)) = 17. \end{aligned}$$

The application of the Tree construct will change the aggregation order and the result might be different with sequential aggregation (such as Reduce). However, below, we will show that if the base workflow satisfies some property, the result of tree aggregation will be equivalent to the one that is obtained by a sequential aggregation.

Definition 4.1. *A workflow $W_a([i_1, \ldots, i_l, \ldots, i_r, \ldots, i_n], o)$ is associative with ports i_l, i_r if and only if it satisfies the associativity law*

$$\begin{aligned} &W_a(p_1, \ldots, W_a(p_1, \ldots, d_1, \ldots, d_2, \ldots, p_{n-2}), \ldots, d_3, \ldots, p_{n-2}) \\ &= W_a(p_1, \ldots, d_1, \ldots, W_a(p_1, \ldots, d_2, \ldots, d_3, \ldots, p_{n-2}), \\ &\quad \ldots, p_{n-2}) \ for\ all\ inputs\ d_1,\ d_2,\ d_3\ and\ p_1, \ldots, p_{n-2}. \end{aligned} \tag{4}$$

Typically, we define that a binary workflow $W_a([i_l, i_r], o)$ is *associative* if and only if it satisfies the *associativity law*

$$\begin{aligned} W_a(W_a(d_1, d_2), d_3) &= W_a(d_1, W_a(d_2, d_3)) \\ &for\ all\ inputs\ d_1, d_2, d_3. \end{aligned} \tag{5}$$

Below, we will show that if the base workflow is associative with the *left tree port* and the *right tree port*, given any list of inputs, the result of the Tree construct-based workflow will be the same as sequential aggregation using the base workflow. We will start the proof from binary workflows.

Theorem 4.2. *Given a binary workflow $W_a([i_1, i_2], o)$ that is associative, for any input list $[d_1, \ldots, d_{2^n}](n \geq 1)$ that can be constructed into a perfect binary tree with depth of n, the unary-construct-based workflow $W_b = T_{i_1,i_2}(W_a)$ satisfies the following equation:*

$$W_b([d_1, \ldots, d_{2^n}]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_{2^n}). \tag{6}$$

Proof.

Basis. Show that the statement holds for $n = 1$. Obviously, we have $W_b([d_1, d_2]) = W_a([d_1, d_2])$, so the statement holds.

Inductive step. Show that if the statement holds for $n = k$, then the statement also holds for $n = k + 1$. This can be done as follows. Assume the equation holds for $n = k$, that is

$$W_b([d_1, \ldots, d_{2^k}]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_{2^k}),$$

and also we can have

$$\begin{aligned} &W_b([d_{2^k+1}, \ldots, d_{2^{k+1}}]) \\ &\quad = W_a(\ldots W_a(W_a(d_{2^k+1}, d_{2^k+2}), d_{2^k+3}) \ldots, d_{2^{k+1}}). \end{aligned}$$

Then, for $n = k + 1$

$$\begin{aligned} &W_b([d_1, \ldots, d_{2^{k+1}}]) \\ &= W_b(W_b([d_1, \ldots, d_{2^k}]), W_b([d_{2^k+1}, \ldots, d_{2^{k+1}}])) \\ &= W_a(W_a(\ldots W_a(W_a(d_1, d_2), d_3), \ldots, d_{2^k}), \\ &\quad W_a(\ldots W_a(W_a(d_{2^k+1}, d_{2^k+2}), d_{2^k+3}), \ldots, d_{2^{k+1}})) \\ &= W_a(W_a(W_a(\ldots W_a(W_a(d_1, d_2), d_3), \ldots, d_{2^k}), \\ &\quad W_a(\ldots W_a(W_a(d_{2^k+1}, d_{2^k+2}), d_{2^k+3}), \ldots, d_{2^{k+1}-1})), \\ &\quad d_{2^{k+1}}) \\ &\quad \ldots \\ &= W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_{2^{k+1}}). \end{aligned}$$

□

Corollary 4.3. *Given a binary workflow $W_a([i_1, i_2], o)$ that is associative, the unary-construct-based workflow $W_b = T_{i_1,i_2}(W_a)$ satisfies the following equation for any input list $[d_1, \ldots, d_n]$:*

$$W_b([d_1, \ldots, d_n]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_n). \tag{7}$$

Proof. Given $W_a([i_1, i_2], o)$ that is associative, we create another workflow $W_c([i_1, i_2], o)$ defined as

$$\begin{aligned} W_c(d_1, d_2) = d_{id}(d_1 = d_{id}, d_2 = d_{id}) \\ or \quad d_2(d_1 = d_{id}, d_2 \neq d_{id}) \\ or \quad d_1(d_1 \neq d_{id}, d_2 = d_{id}) \\ or \quad W_a(d_1, d_2)(d_1 \neq d_{id}, d_2 \neq d_{id}), \end{aligned} \tag{8}$$

where d_{id} is a reserved identity data product of $W_c(d_1, d_2)$ that will not be used in any regular user inputs. We can simply prove that W_c is also *associative* by enumerating all the possibilities (due to the page limit, the proof is skipped).

Obviously, we can replace W_a with W_c without affecting the results for any regular inputs, and vice versa. Then, for any input lists $[d_1, \ldots, d_n]$ $(d_i \neq d_{id})$, the unary-construct-based workflow $W_d = T_{i_1,i_2}(W_c)$ satisfies the following equation:

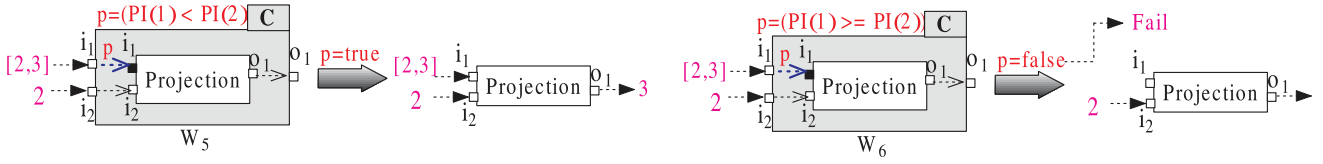


Fig. 8. (a) W_5 created by applying the Conditional construct on the Projection workflow with a predicate $p = (PI(1) < PI(2))$. (b) W_6 created by applying the Conditional construct on the Projection workflow with an opposite predicate $p = (PI(1) >= PI(2))$.

$$\begin{aligned} W_d([d_1,\ldots,d_n]) &= (T_{i_1,i_2}(W_c))([d_1,\ldots,d_n]) \\ &= (T_{i_1,i_2}(W_a))([d_1,\ldots,d_n]) \\ &= W_b([d_1,\ldots,d_n]). \end{aligned} \tag{9}$$

For any input list $[d_1,\ldots,d_n]$ with length n, we increase it to 2^k, where $2^{k-1} < n \leq 2^k$ and fill with identity data products d_{id}. By the definition of the Tree construct, we can derive that

$$W_d([d_1,\ldots,d_n,d_{id},\ldots,d_{id}]) = W_d([d_1,\ldots,d_n]). \tag{10}$$

By theorem 4.2, we have

$$\begin{aligned} &W_d([d_1,\ldots,d_n,d_{id},\ldots,d_{id}]) \\ &\quad = W_c(\ldots W_c(W_c(d_1,d_2),d_3)\ldots,d_{id}) \\ &\quad = W_a(\ldots W_a(W_a(d_1,d_2),d_3)\ldots,d_n). \end{aligned} \tag{11}$$

From (10) and (11), we have

$$W_d([d_1,\ldots,d_n]) = W_a(\ldots W_a(W_a(d_1,d_2),d_3)\ldots,d_n). \tag{12}$$

From (9) and (12), we can conclude (7). □

Corollary 4.3 can be extended to multiport workflows that are associative with two ports since inputs of other ports can be considered as arguments. We will show the proof in the later section with the help of the to-be-proposed Curry construct.

4.4 The Conditional Construct

The Conditional construct enables the conditional execution of a workflow based on a condition on one of the inputs.

As illustrated in Fig. 4d, to apply the Conditional construct on a workflow $W_a([i_1,\ldots,i_n],o)$, one of the input ports of W_a, $i_c \in [i_1,\ldots,i_n]$, is designated as the *conditional port*, on which a logical test will be calculated based on the input data product. A predicate p will be provided by the users as a parameter to evaluate the output of the *conditional port* and W_a can be executed only if p evaluates to be true. p can be modified by the users dynamically and the workflow behavior will, thus, be changed accordingly. The semantics of the Conditional construct $C_{p(i_c)}(W_a([i_1,\ldots,i_n],o))$ can be formulated by the following equation:

$$\begin{aligned} W_b.o &= W_b(i_1,\ldots,i_c,\ldots,i_n) \\ &= p(i_c)?W_a(i_1,\ldots,i_c,\ldots,i_n) : Fail. \end{aligned} \tag{13}$$

Here, the p?A:B notation means exactly *if p then return A else return B*.

For example, given a pair of numbers, W_5 shown in Fig. 8a outputs the second number if it is greater than the first number; otherwise fails. Similarly, W_6 shown in Fig. 8b outputs the second number if it is not greater than the first number; otherwise fail. Both W_5 and W_6 are created from the *Projection* workflow by applying the Conditional construct with input port i_1 designated as the *conditional port*. Because the predicates on W_5 and W_6 are opposite, given the same input, only one of the workflows can be executed and the other one will fail. For instance, given an input pair [2, 3] for the conditional port and a number 2 that is used to specify the number in the pair to be projected, the outputs of W_5 and W_6 are

$$\begin{aligned} W_5.o &= W_5([2,3],2) \\ &= (2<3)?Projection([2,3],2) : Fail \\ &= Projection([2,3],2) = 3, \\ W_6.o &= W_6([2,3],2) \\ &= (2>=3)?Projection([2,3],2) : Fail = Fail. \end{aligned} \tag{14}$$

Traditional *if-then-else* statement and multiple-branch conditional statement can be supported by applying the Conditional construct on different branches of workflows.

4.5 The Loop Construct

The Loop construct enables the cyclic execution of a workflow based on a predicate over the output of the workflow. The output of the workflow will be repetitively returned (fed back) to a specified input port until the predicate evaluates to true. As illustrated in Fig. 4e, to apply the Loop construct on a workflow $W_a([i_1,\ldots,i_n],o)$, one of the input ports of W_a, $i_l \in [i_1,\ldots,i_n]$, is designated as the *loop port*, which takes input either from the initial input data product at the first iteration or the feedback from the output of the previous iteration. A user-input predicate p is set on the output port o such that if p evaluates to false, the output will be fed back to the *loop port*. Therefore, it is required that $dom(W_a.o) \subseteq dom(W_a.i_l)$. The semantics of the Loop construct $L_{i_l,p(o)}(W_a([i_1,\ldots,i_n],o))$ can be formulated by the following recursive equation:

$$\begin{aligned} W_b.o &= W_b(i_1,\ldots,i_l,\ldots,i_n) \\ &= o_m, \\ where\ o_1 &= W_a(i_1,\ldots,i_l,\ldots,i_n) \quad (p(o_1) = false), \\ o_2 &= W_a(i_1,\ldots,o_1,\ldots,i_n) \quad (p(o_2) = false), \\ &\ldots, \\ o_m &= W_a(i_1,\ldots,o_{m-1},\ldots,i_n) \quad (p(o_m) = true). \end{aligned} \tag{15}$$

As an example, the workflow W_7 shown in Fig. 9 repeatedly increase the base value by 1 until it is greater than 100. W_7 is created from a predefined workflow *Add* by applying the Loop construct with input ports i_1 designated as the loop port. Given input list $[0,1]$, the output of W_7 is

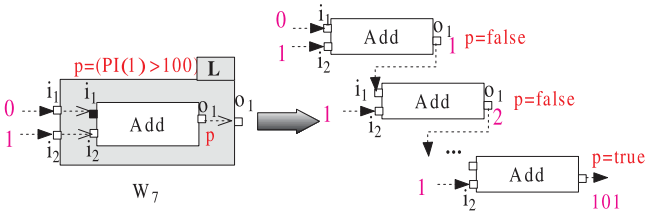


Fig. 9. Workflow W_7 created by applying the Loop construct on an *Add* Workflow.

$$
\begin{aligned}
o_1 &= Add(0,1) = 1 \ (o_1 \not> 100),\\
o_2 &= Add(1,1) = 2(o_2 \not> 100),\\
&\ldots,\\
o_{101} &= Add(100,1) = 101 \ \ (o_{101} > 100),\\
W_7.o &= W_7(0,1) = o_{101} = 101.
\end{aligned}
$$

4.6 The Curry Construct

The Curry construct allows a user to fix one of the input ports with a specified argument and thus reduce the number of input ports. By applying multiple Curry constructs, a workflow that takes multiple arguments can be translated into a chain of workflows each with a single argument.

As illustrated in Fig. 4f, to apply the Curry construct on a workflow $W_a([i_1,\ldots,i_n],o)$, one of the input ports of W_a, $i_u \in [i_1,\ldots,i_n]$ is assigned with an argument. Therefore, the resulted workflow W_b will only have $n-1$ input ports and there are one to one mappings from those ports to the input ports of W_a.

The semantics of the Curry construct $U_{i_u,p}(W_a([i_1,\ldots,i_u,\ldots,i_n],o))$ can be formulated by the following equation:

$$
\begin{aligned}
W_b.o &= W_b(d_1,\ldots,d_{n-1})\\
&= W_a(d_1,\ldots,p,\ldots,d_{n-1}).
\end{aligned}
\tag{16}
$$

As an example, the workflow W_8 shown in Fig. 10 implements the Increment operator by applying The Curry construct on the Addition workflow. W_8 contains only one input port and will automatically increment the input integer by 1.

Theorem 4.4. The Curry construct satisfies *commutativity*

$$
\begin{aligned}
&U_{i_{u_1},p_1}(U_{i_{u_2},p_2}(W_a([i_1,\ldots,i_n],o)))\\
&= U_{i_{u_2},p_2}(U_{i_{u_1},p_1}(W_a([i_1,\ldots,i_n],o)))\\
&\quad (i_{u_1},i_{u_2} \in [i_1,\ldots,i_n], i_{u_1} \neq i_{u_2}).
\end{aligned}
\tag{17}
$$

Proof. Let $W_b = U_{i_{u_2},p_2}(W_a)$ and $W_c = U_{i_{u_1},p_1}(W_b)$.

Then, according to (16), for any given inputs $i_1,\ldots,i_n$, we have

$$
\begin{aligned}
W_c.o &= W_b(i_1,\ldots,p_2,\ldots,i_n)\\
&= W_a(i_1,\ldots,p_1,\ldots,p_2,\ldots,i_n).
\end{aligned}
\tag{18}
$$

Similarly, let $W_d = U_{i_{u_1},p_1}(W_a)$ and $W_e = U_{i_{u_2},p_2}(W_d)$, we can get

$$
\begin{aligned}
W_e.o &= W_d(i_1,\ldots,p_1,\ldots,i_n)\\
&= W_a(i_1,\ldots,p_1,\ldots,p_2,\ldots,i_n).
\end{aligned}
\tag{19}
$$

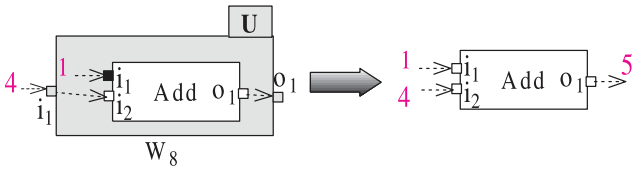


Fig. 10. Workflow W_8 created by applying the Curry construct on an *Add* Workflow.

From (18) and (19), for any given inputs $i_1,\ldots,i_n$, we can conclude (17). □

Theorem 4.5. The Curry construct is *commutative* with all unary constructs

$$
\begin{aligned}
&U_{i_u,p}(M_{i_m}(W_a([i_1,\ldots,i_n],o)))\\
&= M_{i_m}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o)))\\
&\quad (i_u,i_m \in [i_1,\ldots,i_n], i_u \neq i_m),
\end{aligned}
\tag{20}
$$

$$
\begin{aligned}
&U_{i_u,p}(R_{i_b,i_r}(W_a([i_1,\ldots,i_n],o)))\\
&= R_{i_b,i_r}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o)))\\
&\quad (i_u,i_b,i_r \in [i_1,\ldots,i_n], i_u \neq i_b, i_u \neq i_r),
\end{aligned}
\tag{21}
$$

$$
\begin{aligned}
&U_{i_u,p}(T_{i_l,i_r}(W_a([i_1,\ldots,i_n],o)))\\
&= T_{i_l,i_r}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o)))\\
&\quad (i_u,i_l,i_r \in [i_1,\ldots,i_n], i_u \neq i_l, i_u \neq i_r),
\end{aligned}
\tag{22}
$$

$$
\begin{aligned}
&U_{i_u,p}(C_{p(i_c)}(W_a([i_1,\ldots,i_n],o)))\\
&= C_{p(i_c)}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o)))\\
&\quad (i_u,i_c \in [i_1,\ldots,i_n], i_u \neq i_c),
\end{aligned}
\tag{23}
$$

$$
\begin{aligned}
&U_{i_u,p}(L_{i_l,p(o)}(W_a([i_1,\ldots,i_n],o)))\\
&= L_{i_l,p(o)}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o)))\\
&\quad (i_u,i_l \in [i_1,\ldots,i_n], i_u \neq i_l).
\end{aligned}
\tag{24}
$$

Due to the page limit, we will skip those proofs which will be similar to the one for Theorem 4.4. Theorems 4.4 and 4.5 are important foundations that allow a user to set parameters to arbitrary Curry-based workflows and the parameters can be correctly passed to the enclosed base workflow. Below, we prove that Corollary 4.3 can be extended to workflows with multiple input ports.

Theorem 4.6. *Given a workflow $W_a([i_l,\ldots,i_l,\ldots,i_r,\ldots,i_n],o)$ that is associative with ports i_l and i_r, the unary-construct-based workflow $W_b = T_{i_l,i_r}(W_a)$ satisfies the following equation for any inputs $p_1,\ldots,[d_1,\ldots,d_m],\ldots,p_n$:*

$$
\begin{aligned}
&W_b(p_1,\ldots,[d_1,\ldots,d_m],\ldots,p_n)\\
&= W_a(p_1,\ldots W_a(p_1,\ldots,W_a(p_1,\ldots,d_1,\ldots,\\
&\quad d_2,\ldots,p_n),\ldots,d_3,\ldots,p_n)\ldots,d_n,\ldots,p_n).
\end{aligned}
\tag{25}
$$

Proof. We first apply multiple Curry constructs on W_a and each will assign a parameter to one of the input ports except i_l and i_r. W_a can, therefore, be translated to a chain of workflows as follows:

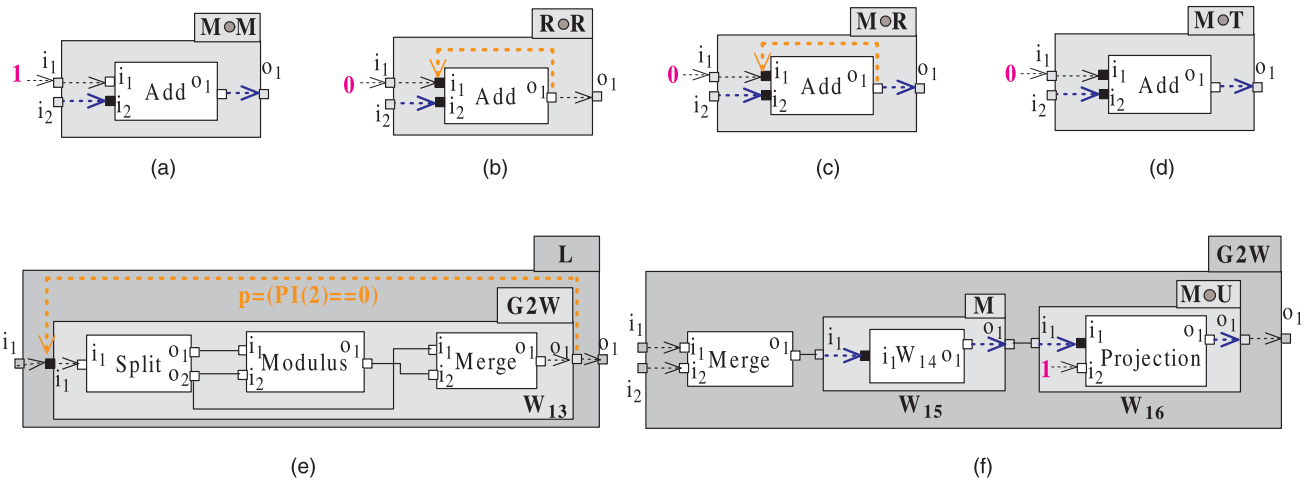


Fig. 11. (a) Unary-construct-based workflow W_9 created by the composition of two Map constructs on the *Add* workflow. (b) Unary-construct-based workflow W_{10} created by the composition of two Reduce constructs on the *Add* workflow. (c) Unary-construct-based workflow W_{11} created by the composition of the Map construct and the Reduce construct on the *Add* workflow. (d) Unary-construct-based workflow W_{12} created by applying the composition of the Map construct and the Tree construct on the *Add* workflow. (e) Unary-construct-based workflow W_{15} created by applying the Loop construct on a graph-based workflow. (f) Graph-based workflow W_{17} created by applying the *G2W* construct on a workflow graph. (a) W_9, (b) W_{10}, (c) W_{11}, (d) W_{12}, (e) W_{14}, (f) W_{17}.

$$\begin{aligned} W_{a_1} &= U_{i_1,p_1}(W_a),\\ W_{a_2} &= U_{i_2,p_2}(W_{a_1}),\\ &\ldots,\\ W_{a_{n-2}} &= U_{i_n,p_n}(W_{a_{n-3}}) = U_{i_n,p_n}(\ldots U_{i_1,p_1}(W_a)\ldots). \end{aligned} \tag{26}$$

By doing this, we can obtain a binary workflow $W_{a_{n-2}}([i_l, i_r], o)$ that is associative. Then, according to Corollary 4.3, the Tree-based workflow $W_{b'} = T_{i_l,i_r}(W_{a_{n-2}})$ satisfies the following equation for any input list $[d_1, \ldots, d_n]$:

$$\begin{aligned} &W_{b'}([d_1,\ldots,d_m])\\ &\quad = W_{a_{n-2}}(\ldots W_{a_{n-2}}(W_{a_{n-2}}(d_1,d_2),d_3)\ldots,d_m). \end{aligned} \tag{27}$$

According to (22), we can get

$$\begin{aligned} W_{b'} &= T_{i_l,i_r}(W_{a_{n-2}})\\ &= U_{i_n,p_n}(T_{i_l,i_r}(W_{a_{n-3}}))\\ &\ldots\\ &= U_{i_n,p_n}(\ldots U_{i_1,p_1}(T_{i_l,i_r}(W_a))\ldots)\\ &= U_{i_n,p_n}(\ldots U_{i_1,p_1}(W_b)\ldots). \end{aligned} \tag{28}$$

Therefore, by (28) and (16) we can get

$$W_b(p_1,\ldots,[d_1,\ldots,d_m],\ldots,p_n) = W_{b'}([d_1,\ldots,d_m]). \tag{29}$$

From (27) and (29), we can get

$$\begin{aligned} &W_b(p_1,\ldots,[d_1,\ldots,d_m],\ldots,p_n)\\ &\quad = W_{a_{n-2}}(\ldots W_{a_{n-2}}(W_{a_{n-2}}(d_1,d_2),d_3)\ldots,d_m). \end{aligned} \tag{30}$$

From (26) and (16), we can get

$$\begin{aligned} &W_{a_{n-2}}(\ldots W_{a_{n-2}}(W_{a_{n-2}}(d_1,d_2),d_3)\ldots,d_m)\\ &\quad = W_a(p_1,\ldots W_a(p_1,\ldots,W_a(p_1,\ldots,d_1,\ldots,\\ &\qquad d_2,\ldots,p_n),\ldots,d_3,\ldots,p_n)\ldots,d_n,\ldots,p_n). \end{aligned} \tag{31}$$

From (30) and (31), we can conclude (25). □

4.7 Workflow Composition

Compared to existing workflow composition models, our model has the following novel characteristics:

1. Workflows are the only operands for workflow composition. All composite workflows are created as the composition of existing workflows.
2. Every workflow can be directly used for workflow composition through workflow constructs and every composition results in a new workflow, either a graph-based workflow or a unary-construct-based workflow.
3. Workflow constructs are fully composable and the set of workflows is closed under all the workflow constructs.

These characteristics make our framework unique in the ability to apply the proposed workflow constructs and their compositions on arbitrary workflows, as illustrated by the following scenarios.

Unary workflow constructs can compose with each other arbitrarily. Given an existing unary-construct-based workflow $W_b = S_1(W_a)$ which is created by applying a unary construct S_1 on a workflow W_a, we can apply another unary construct S_2 on W_b resulting in $W_c = S_2(W_b) = S_2(S_1(W_a))$. We define a composition of S_1 and S_2 as a new unary construct to simplify this two-step composition. The semantics of this new unary construct is given by the following formula:

$$(S_2 \circ S_1)(W_a) = S_2(S_1(W_a)). \tag{32}$$

For example, given a predefined workflow *Add*, applying different compositions of Map and Reduce constructs will result in different workflows. W_9 shown in Fig. 11a is created by applying the composition of two Map constructs. Given inputs of a base value 1 and a table of numbers (represented as a list of lists), W_9 will increase all the numbers in the table by one and output the resulting table. W_{10} shown in Fig. 11b is created by applying the

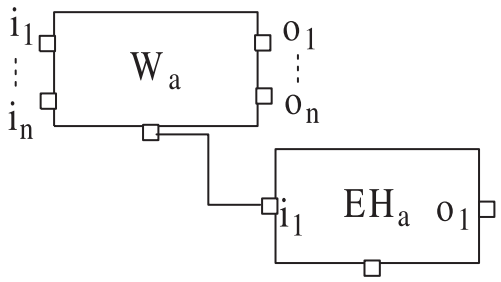


Fig. 12. Workflow exception handling.

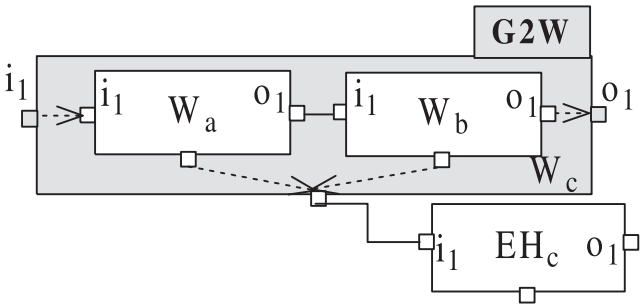


Fig. 13. Workflow exception propagation.

composition of two Reduce constructs. Given inputs of a base value 0 and a table of numbers, W_{10} will output a sum of the whole table. W_{11} shown in Fig. 11c is created by applying the composition of Map and Reduce. Given inputs of a base value 0 and a $m \times n$ table of numbers, W_{11} will output a list of m numbers, each representing the sum of the corresponding row in the table. The composition of unary constructs are arbitrary. Any finite number of application of Map and Reduce constructs is allowed, which enables the processing of data cubes in any dimensions. W_{12} shown in Fig. 11d is created by applying the composition of Map and Tree. Given the same inputs, W_{11} and W_{12} will have the same output. However, W_{12} supports parallel aggregation using the Tree construct.

Unary workflow constructs can compose with other constructs arbitrarily and hierarchically. In particular, our unary workflow constructs can be applied to a graph-based workflow to form a unary-construct-based workflow; several unary-construct-based workflows can also be linked together by data channels to form a workflow graph G and then the $G2W$ construct can be applied to G to form a graph-based workflow.

As an example, W_{14} shown in Fig. 11e implements the euclidean algorithm to calculate the greatest common divisor for a pair of integers. W_{14} is created by applying the Loop construct on a graph-based workflow W_{13}. Given a pair of integers $[a, b]$ as input, W_8 will output a pair of integers $[b, a\%b]$. By the application of the Loop construct, W_{13} will be executed repeatedly until the predicate $p = (PI(2) == 0)$ evaluates to be true, which means that the second number of the pair equals to 0. Finally, W_{14} will output a pair of integers $[gcd(a, b), 0]$, where $gcd(a, b)$ is the greatest common divisor of the input pair. A unary-based workflow W_{15} can then be created by applying the Map construct on W_{14}. W_{15} takes a list of pairs $[[a_1, b_1], \ldots, [a_n, \ldots b_n]]$ and outputs a list of pairs $[[gcd(a_1, b_1), 0], \ldots, [gcd(a_n, b_n), 0]]$.

Further, W_{17} shown in Fig. 11f can process two lists in parallel and calculate the greatest common divisor for each corresponding pair in two lists. Given two lists $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$ as input, W_{17} will output a list $[gcd(a_1, b_1), \ldots, gcd(a_n, b_n)]$ containing the greatest common divisors for each corresponding pair. W_{17} is created by applying the $G2W$ construct on a workflow graph that contains three workflows. The two input lists are merged into one list of pairs $[[a_1, b_1], \ldots, [a_n, \ldots b_n]]$ by the *Merge* workflow and sent to W_{15}. W_{15} then outputs a list of pairs $[[gcd(a_1, b_1), 0], \ldots, [gcd(a_n, b_n), 0]]$ to W_{16}, which is created by applying the Map construct and Curry construct on a $Projection$ workflow. W_{16} will project the first element in each pair resulting in a list $[gcd(a_1, b_1), \ldots, gcd(a_n, b_n)]$.

5 A Dataflow-Based Approach for Exception Handling

Much research has been carried out on issues of exception handling in workflow management systems [20], [21], [22], [25], [26]. However, most of the existing approaches are rule or event based. In this section, we propose a dataflow-based approach for exception handling that is compatible with our scientific workflow composition model.

5.1 Exception Handling

In our approach, an exception is represented as a special data product (called *exception data product*) that contains exception information. As shown in Fig. 12, each workflow contains a default exception port as the output port specifically for an exception data product. The exception port can be linked to exception handling workflows such as Stop, Pause, and user defined handlers.

As the basic building block, a primitive workflow is responsible to capture all the exceptions during the invocation of inside tasks, generate corresponding workflow exception data products, and output through the exception port. Workflow exceptions can also be propagated hierarchically to higher level composite workflows following the workflow construction. As shown in Fig. 13, the exception ports of W_a or W_b are automatically mapped to the exception port of W_c. Therefore, whenever an exception data product e_1 is generated by either W_a or W_b, it will be passed to W_c and W_c will generate a new exception data product e_2 that contains e_1 as well as the information of W_c.

5.2 The Exception Construct

The data exception construct enables a user to capture the data exception on one of the input/output ports. As illustrated in Fig. 14, to apply the Exception construct on a workflow $W_a([i_1, \ldots, i_n], o)$, one of the input/output ports is designated as the *exception test port*, on which a logical test will be calculated based on the input/output data product. A user-input predicate p is set on the *exception test port* and a user-defined exception data product e needs to be designated to the exception port. If p evaluates to be true, W_b will behave exactly as W_a, otherwise, W_b will output e from the exception port.

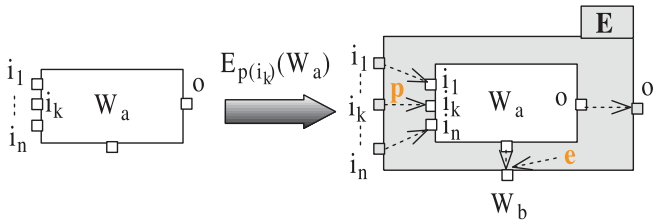


Fig. 14. The exception construct.

As an example shown in Fig. 15, W_{18} detects the typical division by zero error and outputs an exception data product.

6 Case Studies

The proposed techniques have been realized in an XML-based scientific workflow specification language, called WSL [27], and implemented in a new version of the VIEWsystem [9]. The implementation details are out of the scope of this paper. However, we will present several case studies in order to validate our techniques.

6.1 Workflow for Freebase Processing

We implemented a *Freebase Processing* Workflow as shown in Fig. 16 to validate the ability of our model to leverage MapReduce tasks to the workflow level. We chose Amazon Elastic MapReduce [28] for this case study. Amazon Elastic MapReduce is a Web service that utilizes a hosted Hadoop framework running on the web-scale infrastructure. Amazon Elastic MapReduce published three sample job flows [29] that are used to filter a set of Freebase data and store it into Amazon SimpleDB data store. In our experiment, we created three primitive workflows: WFreequentID, WDataStorage, and WNameStorage, and each is based on one of the job flows. The WFreequentID workflow can iterate over each file of input to look for the most popular Freebase IDs. The WDataStorage workflow stores the results of the WFreequentID workflow into Amazon SimpleDB. The WNameStorage workflow reads Freebase data and stores names and their IDs into Amazon SimpleDB. We then created a graph-based Freebase Processing Workflow that is composed by those three workflows. Our workflow technique automatically connects the execution of the three workflows via explicit dataflows, and naturally enables concurrent execution of the WFreequentID workflow and the WNameStorage workflow as they do not have data dependencies.

6.2 Workflows for Matrix Summation

We designed two workflows for matrix summation in order to validate the performance of our Map construct. The first

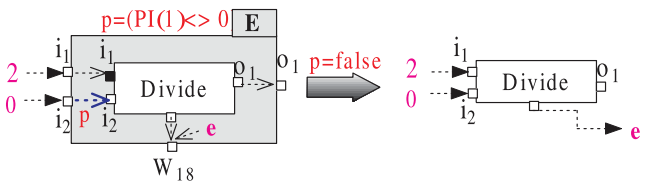


Fig. 15. Workflow W_{18} created by applying the Exception construct on a *Divide* workflow.

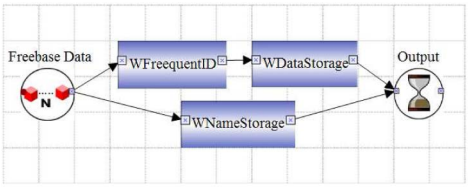


Fig. 16. The *Freebase Processing* Workflow.

workflow sequentially adds up all the elements in the matrix. The second workflow takes advantage of the Map construct and calculates the sum of each row in parallel, and then sums up the results using a Reduce-based addition workflow. We add a delay of 10 ms for each addition task for better observation and run the two workflows on 10 matrices with different sizes. Fig. 17 compares the performance of the two workflows. The efficiency of the first workflow is $O(n^2)$ while the efficiency of the second workflow is $O(n)$ (n represents the dimension of the matrix).

6.3 Workflow for Biological Simulation

The marine worm *Nereis succinea* spawns during a coordinated "nuptial dance" timed by the phases of the moon, initiated by the time of the day, and choreographed by the exchange of chemical signals [30]. Females excrete a pheromone that can be attractants for the opposite sex in many environments. We developed a simulation model of male and female *N. succinea* behavior [31] for testing the hypothesis that male responses to low concentrations of CSSG can facilitate finding females. Recently, we designed a Simulation Analysis Workflow using the VIEWsystem as shown in Fig. 18.

The *Simulation* workflow shown in Fig. 18a takes a list of simulation parameters (SP) and the number of simulations (N) as inputs, and outputs the number of successful matings. SP contains a list of parameters such as ((condition, 8), (starting-Angle, 30)). A *Repeat* workflow with an application of the Conditional construct generates an output list of N copies of SP, provided that SP satisfies conditions such as "$(PI(condition) < 11)\&\&(PI(condition) > 6)$." The *Single Simulation Web Service* (SSWS) workflow is a primitive workflow created from a Web service that can run a single simulation and generate a Boolean mating result.

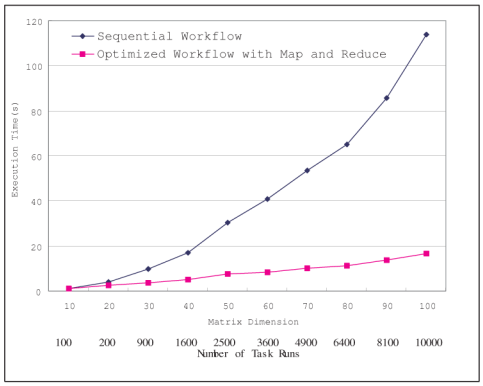


Fig. 17. Performance comparison of two workflows for matrix summation.

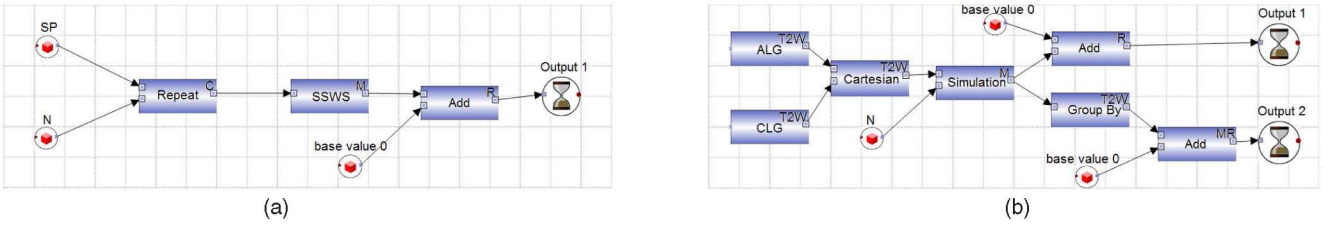


Fig. 18. (a) The *Simulation* workflow. (b) The *Analysis* workflow.

The application of the Map construct allows *SSWS* to run multiple times simultaneously and generate a result list containing the mating results, which will be summed up by a unary-construct-based workflow created by applying the Reduce construct on an *Add* workflow.

The *Analysis* workflow shown in Fig. 18b is designed for testing male behaviors under different concentrations with different starting angles. The *Angle List Generator* (ALG) workflow and the *Concentration List Generator* (CLG) workflow allow a user to set the testing parameters and generate a list of *starting-angle* parameters and a list of *concentration* parameters, respectively. The *Cartesian* workflow will do a cartesian join on those two lists and produce a list of parameter sets containing all the combinations of these two parameters. A *Simulation* workflow with an application of the Map construct can process all the parameter sets in parallel and produce a list of integers, each representing the number of successful matings for the corresponding simulation. Then, an *Add* workflow with an application of the Reduce construct will iteratively add up the successful mating numbers in the list and calculate the sum of total successful matings. However, domain scientists are also particularly interested in the behaviors for each chemical concentration. Therefore, we use a *Group By* workflow to group the results into a group of lists with each list containing the successful mating numbers for a given concentration. Another *Add* workflow with an application of a composition of Map and Reduce will generate a list of numbers, each of which representing a subtotal of the successful mating numbers with different starting-angles under a certain concentration.

7 Related Work

Much research has been done in business workflow languages such as BPEL [2], [26], a standard business process execution language, and YAWL [3], a formal workflow language based on Petri Nets. However, most of these languages are controlflow oriented as business workflows are driven by business rules and it is important to maintain the state of a business process and to provide controlflow constructs to formulate state-based business rules. Although some constructs, such as *ForEach*, *If*, *While* in BPEL 2.0 [26]; *MultipleInstance*, *Structured Loop*, *Multiple Choice*, and *Parallel Split* in YAWL [3], have been proposed for business workflows to support iteration and concurrency, they cannot be directly applied to a dataflow-based scientific workflow composition framework due to the fundamental differences between controlflow and dataflow. For example, in contrast to the Map construct, which returns a list of results, the *ForEach* construct returns nothing (since it is a controlflow construct). Considering the dataflow-oriented nature, the Map construct is more natural for scientific workflows as the results can be directly fed to the input of subsequent workflows or tasks. Similarly, Wong and Gibbons [32] proposed a process-algebraic approach to model workflows as CSP processes and support various controlflow patterns. OWL-S [33] provides *Split+Join* for the parallel execution of Semantic Web services while the Web Service Modeling Ontology (WSMO) [34] also supports parallel workflows through a set of controlflow-based transition rules that are executed in parallel.

Recently, data-centric approaches have received much recognition to model medium or large sized business workflows. Frits et al. [35] introduced an artifact-centric approach, which focuses on recording "business artifacts" including business objects, their life cycles, and provenance information. E-BioFlow [36], a workflow system built on top on YAWL [3], provides three perspectives (controlflow, dataflow, and resource) to support workflow design. The information of the three perspectives will all be translated to controlflows during runtime. However, in essence, these approaches are controlflow-based rather than dataflow-based.

The experience of business workflows demonstrated the importance of developing formal models, languages, and tools to support workflow modeling, analysis, verification, and communication [8]. However, while business workflows focus mostly on the correctness and reliability of business processes; scientific workflows focus more on efficiency and scalability. Slominski [37] categorized several requirements for scientific workflows, some of which concur with ours, such as the requirement that a workflow engine should support hierarchical composition to encourage workflow reuse. Zhao et al. [38] identified the importance of the separation of the logical data structure from its physical representation and introduced an XDTM model to define the physical and logical structures as well as their mappings. Deelman [39] discussed the future challenges in workflow languages and representations and pointed out that a workflow language should be simple and focus on workflow composition rather than being a full-fledged programming or scripting language. Visual programming is still a challenge for current workflow design tools when a workflow involves thousands of tasks.

Although several dataflow-based scientific workflow management systems (SWFMSs) have been developed [4], [5], [6] and most of them implement a proprietary scientific workflow language, research on scientific workflow models and languages is very limited. Kepler [4] implements a scientific workflow language, called MoML, and provides

an *IteratorOverArray* actor (Kepler's term of workflow task) to support iterated execution. However, this actor does not directly support parallel execution of its contained actor. Taverna [5] implements a scientific workflow language, called Scufl, and provides implicit iteration by allowing a user to specify the iteration strategy of each processor (Taverna's term of workflow task). Taverna can simulate control links using data links [40] and support If-Else behavior by using control links and two distinguished processors called "Fail-if-false" and "Fail-if-true." MOTEUR [41] supports both the parallel processing of independent data with a single service on different computing resources (called "data parallelism") and the parallel execution of different services with different data sets (called "services parallelism"). However, none of those models supports the composition of constructs. VisTrails [6] features an action-based mechanism to automatically capture workflow evolution provenance. However, VisTrails does not provide the workflow constructs proposed in this paper.

A few other controlflow-based scientific workflow languages have been proposed [42], [43], [44], [45]. Pegasus [42] aims to take advantage of Grids for parallel processing at the task level and its workflow language DAX can describe controlflow-based sequential and parallel workflows. Swift [43] implements a C-like language SwiftScript and provides the *foreach* construct that is similar to those in business workflow languages. Triana [44] provides a pair of constructs *AND-split* and *AND-join* for controlflow-based parallel execution. Martlet [45] is a scientific workflow language that supports MapReduce-style workflows. However, because it is controlflow-based, the constructs introduced in Martlet are inapplicable to dataflow-based scientific workflows in which input ports and output ports are well defined. Moreover, the composability of Martlet is very limited as Martlet constructs cannot be applied in a nested way.

None of the above workflow languages provides the constructs that can be applied on arbitrary workflows. In contrast, our constructs can be applied on arbitrary workflows. Furthermore, we can also apply the composition of multiple constructs on a workflow because applying a construct on a workflow will result in another workflow on which another construct can be applied.

This paper provides a holistic framework for scientific workflow composition, covering requirement analysis, composition model, implementation, and evaluation. This paper extends [27] with the following additional contributions:

1. We identify seven key requirements for a scientific workflow composition model in a new Section 2.
2. We introduce two new constructs Tree and Curry in Section 4. We also identified and proved several properties for the constructs.
3. We propose a dataflow-based exception handling approach, resulting in a new Section 5.
4. Two more case studies are presented in Section 6 to validate our techniques.

8 Conclusions and Future Work

In this paper, We proposed a dataflow-based scientific workflow composition model in which workflow constructs are fully compositional one with another. We implemented the proposed framework in a new version of our VIEW system and conducted several case studies to validate our proposed techniques. In the future, we plan to investigate the fundamental differences between dataflow and controlflow, and study the feasibility of mapping our work to the Petri Nets based YAWL workflow language. We also plan to enhance our work with Cloud computing [46] techniques to improve the performance and scalability of our framework.

Acknowledgments

The authors would like to thank other members of the VIEW team, particularly Cui Lin, for their helpful discussions and close collaboration in the development of the VIEW system. The authors would also give thanks to Jeffery L. Ram for his collaborative work on the TangoInSilico biological simulation project, which is used for the case study presented in this paper.

References

[1] C. Lin, S. Lu, X. Fei, A. Chebotko, Z. Lai, D. Pai, F. Fotouhi, and J. Hua, "A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution," *IEEE Trans. Services Computing*, vol. 2, no. 1, pp. 79-92, Jan.-Mar. 2009.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business Process Execution Language for Web Services, Version 1.1," http://www.ibm.com/developerworks/library/specification/ws-bpel, 2003.

[3] W. van der Aalst and A. ter Hofstede, "YAWL: Yet Another Workflow Language," *Information Systems*, vol. 30, no. 4, pp. 245-275, 2005.

[4] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039-1065, 2006.

[5] T. Oinn, M.J. Addis, J. Ferris, D. Marvin, M. Senger, T. Carver, M. Greenwood, K. Glover, M. Pocock, A. Wipat, and P. Li, "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045-3054, 2004.

[6] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo, "VisTrails: Visualization Meets Data Management," *Proc. ACM Int'l Conf. Management of data SIGMOD*, pp. 745-747, 2006.

[7] I. Taylor, E. Deelman, D. Gannon, and M. Shields, *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag, 2007.

[8] W. van der Aalst and K. van Hee, *Workflow Management: Models, Methods, and Systems*. MIT, 2002.

[9] C. Lin, S. Lu, Z. Lai, A. Chebotko, X. Fei, J. Hua, and F. Fotouhi, "Service-Oriented Architecture for VIEW: A Visual Scientific Workflow Management System," *Proc. IEEE Int'l Conf. Services Computing (SCC)*, pp. 335-342, 2008.

[10] F. DeRemer and H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," *Proc. Fachtagung über Programmiersprachen*, pp. 80-89, 1976.

[11] M. Gorlick and A. Quilici, "Visual Programming-in-the-Large versus Visual Programming-in-the-Small," *Proc. IEEE Symp. Visual Languages*, pp. 137-144, 1994.

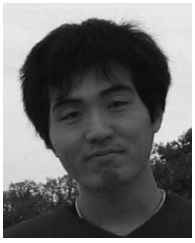
[12] W. Johnston, J. Hanna, and R. Millar, "Advances in Dataflow Programming Languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1-34, 2004.

[13] S. Bowers, B. Ludäscher, A. Ngu, and T. Critchlow, "Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow," *Proc. 22nd Int'l Conf. Data Eng. Workshops*, vol. 0, p. 70, 2006.

[14] Y. Simmhan, B. Plale, and D. Gannon, "Karma2: Provenance Management for Data-Driven Workflows." *Int'l J. of Web Services Research*, vol. 5, no. 2, pp. 1-22, 2008.

[15] S. Lu and J. Zhang, "Collaborative Scientific Workflows," *Proc. IEEE Int'l Conf. in Web Services (ICWS)*, pp. 527-534, 2009.

[16] L. Moreau, J. Freire, J. Futrelle, R. Mcgrath, J. Myers, and P. Paulson, "The Open Provenance Model: An Overview," *Proc. Provenance and Annotation of Data and Processes,* pp. 323-326, 2008.
[17] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe, "Wide Area Data Replication for Scientific Collaborations," *Proc. IEEE/ACM Sixth Int'l Workshop Grid Computing,* pp. 1-8, 2005.
[18] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt, and G. Heber, "Scientific Data Management in the Coming Decade," *ACM SIGMOD Record,* vol. 34, no. 4, pp. 34-41, 2005.
[19] E. Deelman and A. Chervenak, "Data Management Challenges of Data-Intensive Scientific Workflows," *Proc. IEEE Eighth Int'l Symp. Cluster Computing and the Grid (CCGRID),* pp. 687-692, 2008.
[20] M. Adams, A. ter Hofstede, D. Edmond, and W. van der Aalst, "Facilitating Flexibility and Dynamic Exception Handling in Workflows through Worklets," *Proc. 17th Int'l Conf. Advanced Information Systems Eng. (CAiSE),* pp. 45-50, 2005.
[21] N. Russell, W. van der Aalst, and A. ter Hofstede, "Workflow Exception Patterns," *Proc. Advanced Information Systems Eng. (CAiSE),* pp. 288-302, 2006.
[22] C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," *IEEE Trans. Software Eng.,* vol. 26, no. 10, pp. 943-958, Oct. 2000.
[23] C. Lin, S. Lu, X. Fei, D. Pai, and D. Hua, "A Task Abstraction and Mapping Approach to the Shimming Problem in Scientific Workflows," *Proc. IEEE Int'l Conf. Services Computing (SCC),* pp. 284-291, 2009.
[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Conf. Opearting Systems Design and Implementation,* pp. 137-150, 2004.
[25] D. Shukla and B. Schmidt, *Essential Windows Workflow Foundation.* Addison-Wesley Pearson Education, 2007.
[26] OASIS, "Web Services Business Process Execution Language Version 2.0," http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2011.
[27] X. Fei, S. Lu, and C. Lin, "A MapReduce-Enabled Scientific Workflow Composition Framework," *Proc. IEEE Int'l Conf. Web Services (ICWS),* pp. 663-670, 2009.
[28] "Amazon Elastic MapReduce," http://aws.amazon.com/elasticmapreduce, 2011.
[29] "Introduction to Amazon Elastic MapReduce," http://awsmedia.s3.amazonaws.com/pdf/introduction-to-amazon-elastic-mapreduce.pdf, 2011.
[30] J. Ram, C. Müller, M. Beckmann, and J. Hardege, "The Spawning Pheromone Cysteine-Glutathione Disulfide ('Nereithione') Arouses a Momponent Nuptial Behaviour and Electrophysiological Activity in Nereis Succinea Males," *J. Official Publication of the Federation of Am. Soc. for Experimental Biology (FASEB),* vol. 13, pp. 945-952, 1999.
[31] X. Fei, S. Lu, T. Breithaupt, J. Hardege, and J. Ram, "Modeling Matefinding Behavior of the Swarming Polychaete, Nereis Succinea, with TangoInSilico, a Scientific Workflow Based Simulation System for Sexual Searching," *Invertebrate Reproduction and Development,* vol. 52, nos. 1/2, pp. 69-80, 2008.
[32] P. Wong and J. Gibbons, "A Process-Algebraic Approach to Workflow Specification and Refinement," *Proc. Sixth Int'l Conf. Software Composition,* pp. 51-65, 2007.
[33] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "OWL-S: Semantic Markup for Web Services," http://www.w3.org/Submission/OWL-S, 2011.
[34] D. Roman, H. Lausen, U. Keller, U. Oren, C. Bussler, M. Kifer, and D. Fensel, "Web Service Modeling Ontology (WSMO)," http://www.wsmo.org/2004/d2/v1.0/, 2011.
[35] C. Fritz, R. Hull, and J. Su, "Automatic Construction of Simple Artifact-Based Business Processes," *Proc. 12th Int'l Conf. Database Theory (ICDT),* pp. 225-238, 2009.
[36] I. Wassink, H. Rauwerda, P. van der Vet, T. Breit, and A. Nijholt, "E-BioFlow: Different Perspectives on Scientific Workflows," *Proc. Bioinformatics Research and Development (BIRD),* pp. 243-257, 2008.
[37] A. Slominski, "Adapting BPEL to Scientific Workflows," *Proc. Workflows for e-Science: Scientific Workflows for Grids,* pp. 208-226, 2007.
[38] Y. Zhao, M. Wilde, and I. Foster, "Virtual Data Language: A Typed Workflow Notation for Diversely Structured Scientific Data," *Proc. Workflows for e-Science: Scientific Workflows for Grids,* pp. 258-275, 2007.
[39] E. Deelman, "Looking into the Future of Workflows: The Challenges Ahead," *Workflows for e-Science: Scientific Workflows for Grids,* 2007.
[40] D. Turi, P. Missier, C. Goble, D. Roure, and T. Oinn, "Taverna Workflows: Syntax and Semantics," *Proc. IEEE Third Int'l Conf. e-Science and Grid Computing,* pp. 441-448, 2007.
[41] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR," *Int'l J. High Performance Computing Applications,* vol. 22, no. 3, pp. 347-360, 2008.
[42] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz, "Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems," *J. Scientific Programming,* vol. 13, no. 3, pp. 219-237, 2005.
[43] Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde, "A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data," *ACM SIGMOD Record,* vol. 34, no. 3, pp. 37-43, 2005.
[44] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming Scientific and Distributed Workflow with Triana Services," *Concurrency and Computation: Practice and Experience,* vol. 18, no. 10, pp. 1021-1037, 2006.
[45] D. Goodman, "Introduction and Evaluation of Martlet: A Scientific Workflow Language for Abstracted Parallelisation," *Proc. 16th Int'l Conf. World Wide Web,* pp. 983-992, 2007.
[46] L. Zhang and Q. Zhou, "CCOA: Cloud Computing Open Architecture," *Proc. IEEE Int'l Conf. Web Services (ICWS),* pp. 607-616, 2009.



Xubo Fei is currently working toward the PhD degree in the Department of Computer Science, Wayne State University. He is currently a member of the Scientific Workflow Research Laboratory (the SWR Lab). His current research interests include scientific workflows, cloud computing and their applications in bioinformatics, and biology simulation. He is a student member of the IEEE.

Shiyong Lu received the PhD degree in computer science from the State University of New York at Stony Brook in 2002. He is an associate professor in the Department of Computer Science at Wayne State University and the director of the Scientific Workflow Research Laboratory (SWR Lab). His current research interests focus on scientific workflows and their applications. He is an author of two books and more than 80 articles published in various international journals and conferences, including *IEEE Transactions on Services Computing (TSC)*, *Data and Knowledge Engineering (DKE)*, and *IEEE Transactions on Knowledge and Data Engineering (TKDE)*. He is the founding chair of the IEEE International Workshop on Scientific Workflows (SWF) and a founding editorial board member of both the *International Journal on Semantic Web and Information Systems* and the *International Journal of Healthcare Information Systems and Informatics*. He is a senior member of the IEEE.